A Reverse Inheritance Relationship for Improving Reusability and Evolution: The Point of View of Feature Factorization

Ciprian-Bogdan Chirilă, Pierre Crescenzo, Philippe Lahire

A Reverse Inheritance Relationship for Improving Reusability and Evolution: The Point of View of Feature Factorization

Ciprian-Bogdan Chirila[*], Pierre Crescenzo[**], and Philippe Lahire[**]

[*] University Politehnica of Timişoara, Romania,
chirila@cs.utt.ro
[**] University of Nice-Sophia Antipolis, France
Pierre.Crescenzo@unice.fr, Philippe.Lahire@unice.fr

ABSTRACT

Inheritance is one important and controversial issue of object-oriented programming, because of it's different implementations and domain uses: design methods, database, knowledge representation, data mining, object programming languages, modelling ...

Most of the object-oriented programming languages have a direct implementation of specialization, thus we promote the idea that a relationship between classes based on generalization can help in the process of reuse, adaptation, limited evolution of class hierarchies. We name it *reverse inheritance*.

Our goal is to show that reverse inheritance class relationship and it's supporting mechanisms can be used to accomplish the objectives mentioned earlier. Another goal is to prove the feasibility of the approach. On the other hand we analyze some use cases on how the objectives are reached.

Keywords: reverse inheritance, factoring, reuse, adaptation, limited evolution

1. INTRODUCTION

Inheritance is the mechanism used in object-oriented languages to specialize and to adapt the behaviour of a class. It is the backbone of any object-oriented system. As many implementations for inheritance exist, as many object-oriented programming paradigms[3] may be considered.

Inheritance notably offers a way to share (to factor) common features (attributes and methods) between classes, leading to hierarchies without multiple declarations of the same feature.[2]

In our approach we propose to use *reverse inheritance* relationship between classes to improve software reusability (to adapt it according to the context) and to address small evolution or refactoring. Reverse inheritance can be particularly useful when we want to reuse class hierarchies that are developed independently in different contexts. Following this idea, we would like to use reverse inheritance in order to implement a limited way to perform separation of concerns. It is important to note that we address in this paper reverse inheritance in the framework of a language which supports only single inheritance like Java. Impact of multiple inheritance and assertions will be studied but they are out of the scope of this paper.

One of our previous reports[1] proposes a set of features which should be associated to reverse inheritance in order to address the objectives mentioned above. These features are dealing with the insertion of new methods or attributes, their factorization, renaming or redefinition and the access to the code of the descendant. Each of these features must be studied into details and this paper is a first attempt for the description of the main issues related to factorization and renaming. This report addresses also the main uses of reverse inheritance such as inserting a class into a hierarchy, to link two hierarchies, etc. Moreover it describes another use of reverse inheritance which is mainly related to the specification of class hierarchy refactoring. This implied that the use of reverse inheritance is volatile and that the only relationships that persists are inheritance relationships. Objectives of this paper show that it is not the type of uses that we address from now.

The paper is organized in the following way: The second section presents some of the main aspects of the factoring mechanisms according to the state of the art. In the third section we propose a possible syntax and

implementation directions are provided for the factoring mechanism. The fourth section addresses more especially the handling of signature matching and adaptation (syntax and feasibility). Section five draws the conclusions of our study and states the future works.

2. TOWARDS THE DEFINITION OF THE FACTORING MECHANISM

As it has been said in the introduction, we focus on the *factoring mechanism* which works along with other supporting mechanisms like: feature adding, descendant access or renaming.[1] The factoring mechanism in the context of reverse inheritance class relationship relies on the relocation of methods and attributes from classes to their superclasses.

There are many reasons for factoring a class hierarchy: i) multiple occurrences of a method along the inheritance tree means an overhead to the calling mechanism because of the name resolution conflict[2]; ii) multiple declared fields along the inheritance path induce multiple redundant modifications of its occurrences[2]; iii) it is more natural to define concrete subclasses and then to extract commonalities into superclasses.[6]

Sakkinen [7] discusses Pedersen's approach of factorization, which involves the factorization of features from one selected, principal subclass. He proposes that the programmer should specify for each method from which subclass it should be factored.

Moreover [4] defines the features which are common to a set of classes: these features must have the same name in each class or are subject to be renamed. With this approach it is possible to define a signature to which corresponding signatures in each class must conform. This signature may also contain precondition (respectively postcondition) that must not be weaker (respectively stronger) than the ones associated to the methods to be factored. For the common features also, it should be possible to define a precondition other than False which is not weaker than the precondition for the feature in each class *.

In [2] the authors analyze algorithms, based on Galois sub-hierarchies. They are applied to hierarchies in order to find an ideal factorization from the point of view of minimizing the number of feature declarations. They use metrics to count the number of occurrences of redundant features and propose algorithms for restructuring in order to build an optimal hierarchy.

In the approach of [5] which deal with refactoring, they propose a factorization methodology which modifies the code but preserves it's original behaviour. The methodology consists in isolating common features and code, and creating abstract superclasses, based on the following steps: i) to add function signatures to superclasses, ii) to make function bodies compatible, iii) to move variables and to migrate common behaviour to the abstract superclass.

In our approach the reverse inheritance relationship is used as a mean to increase reusability, so that it is close from Pedersen and Sakkinen approaches that integrates it as a basic language mechanism. We consider that features have to be factored in the following manner: common attributes have to be moved from subclasses into superclasses and common signatures of methods have to be copied into superclasses, creating new abstract methods †. Our approach compared with [2] is not automatic. We explore also the possibility to adapt (when it is meaningful) the signature of non-matching features when they have the same semantics (see section 4). In figure 1 the two classes contain a common attribute (*attribute1*) and a common method (*method1()*) which are factored. For the reverse inheritance class relationship we proposed the use of a new keyword *infers* in the definition of the new superclass. Situations like the one presented in our example, with factored features having the same signatures, are quite rare and context dependant. Real situations dealing with method having slightly different signatures but that should be factored must be analyzed and solved. In our approach, the problems related to factorization which are discussed in the next sections are the following: i) how to identify the feature that should be factored - signature matching and, ii) how to adapt these features in order to match the signature specified within the superclass - signature adaptation.

*This approach is related to Eiffel.

†The impact of access modifiers like *public*, *protected* or *private* is not discussed in this paper but it is taken into account by the approach.

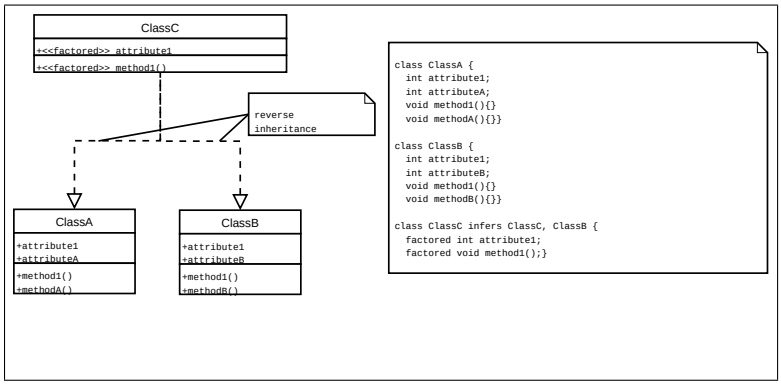

Figure 1. Reverse inheritance factoring mechanism

3. SYNTAX AND IMPLEMENTATION OF FACTORING MECHANISM

In this section we discuss the language syntax and the implementation of the factoring mechanism. In order to show the feasibility of the factoring mechanism we decided to use code transformations to eliminate the reverse inheritance class relationship and to build equivalent hierarchies using just inheritance relationship. We will generate internally equivalent pure Java code and this code does not intend to be shown to the programmer. We propose to analyze examples of the two main situations where reverse inheritance may be involved (single and multiple reverse inheritance). Hierarchy 1 of figure 2 shows an example of single reverse inheritance. The

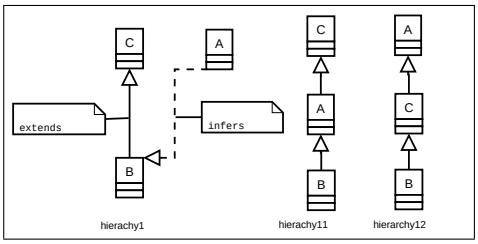

Figure 2. Implementation 1

equivalent *hierarchy11* class diagram can be used when we want to add, to abstract, to redefine, to rename methods in class *B*. The *hierarchy12* from figure 2 can be used when we need to affect in the same way features from class *B* and inherited features from class *C*.

Proposed Java Syntax We propose the following language extension constructions that illustrate single reverse inheritance and fit the context corresponding to the structure of the *hierarchy1* class diagram:

```
class C {}
class B extends C {
  void feature() {/* implementation */}
  void future_factored_feature() {/* implementation */}
  void future_renamed_feature() {/* implementation */}}
class A infers B {
  int new_attribute;
  void new_method(){}
  factored void future_factored_feature();
  void renamed_feature()={void B.future_renamed_feature()};}
```

For the abstraction of features the capability to be used is the factoring mechanism. Abstracted features are declared using the *factored* keyword.

Another situation, a more complex one, is presented in hierarchy 2 of figure 3; it deals with multiple reverse inheritance.

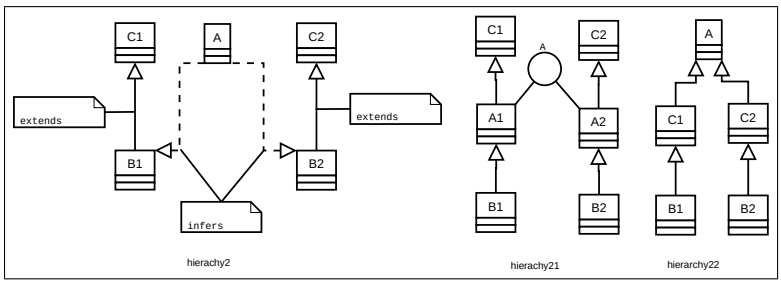

Figure 3. Implementation 2

Like it has been made for single reverse inheritance, we propose two possible implementations depending on the features to be factored. The first implementation solution *hierarchy21* will be used to affect classes *B1* and *B2*. It has two intermediate classes *A1* and *A2* between *C1* and *B1*, respectively *C2* and *B2*. On the other hand an interface *A* is added, which is implemented by the new inserted classes *A1* and *A2* [‡]. Interface *A* will be used to record any possible abstracted features from it's implementing classes *A1* and *A2*. The second implementation solution proposed in *hierarchy22* may be used when we want to affect classes *C1* and *C2*. It has only one superclass A that will include all the factored features and all the new added features. This implementation may be used when we need to factor also inherited features from classes *C1* and *C2*.

4. SIGNATURE MATCHING AND ADAPTATION

In this section we present aspects dealing with signature matching and adaptation in the framework of method factorization. It is necessary to set the rules which define the methods (to be factored) from subclasses that may match the method signature within the superclass and to adapt their signature when it is needed. This may lead to type conversion and to some method renaming. To adapt the signature of methods of subclasses when they are factored enables to make them conform to the signature of the superclass method. This is quite useful because it extends the expressiveness of the factorization mechanism in order to apply it to more methods.

All the entities that contribute to the definition of a signature, are involved in the feature lookup: return type, method name, number of parameter, name, type, position and default value (when available) of each parameter, assertions such as preconditions and postconditions (available in Eiffel)[§].

The work described in [8] involves only signature matching which is based strictly on type analysis. Several cases of possible signature matches are mentioned: exact match, partial relaxed match, transformation relaxed match, combined relaxed match, generic match. Cases of relaxed match, where types are substituted with conforming ones, will be used in our study to reach the goals mentioned earlier.

Potential solutions for name matching, which link concrete methods from subclasses with the correspondent abstract ones in the superclass, are: the use of annotations (a set of meta-information written by the programmer in the source code), information that can be extracted from comments, manual setting of the factored features.

In our approach we address the latter solution and we extend the syntax of Java in order to improve the expressiveness of reverse inheritance class relationship. A sample written in the Java language extension has the following flavour:

[‡]According to the integration of an adding mechanism with reverse inheritance relationship, classes *A1* and *A2* could be used for storing the new added features.

[§]Assertion handling is out of the scope of the paper.

```
01 class Parallelogram {
02   void paint(Canvas c, int x, int y) {/* parallelogram implementation */}}
03 class Ellipse {
04   void update(double x, double y, Canvas canvas) {/* ellipse implementation */}}
05 class Shape infers Parallelogram, Ellipse {
06   factored void paint(Canvas c, int x, int y)= {
07     Parallelogram.paint(Canvas c, int x, int y),
08     Ellipse.update(double x -> x, double y -> y, Canvas canvas -> c)};}
```

Line *05* introduces a new keyword *infers* for expressing the reverse inheritance between class *Shape* and classes *Parallelogram* and *Ellipse*. Between lines *06* and *08*, the common painting method is factored *void paint(Canvas c,int x, int y)*, which corresponds to method *paint(...)* from class *Parallelogram* and *update(...)* from class *Ellipse*. The *factored* keyword is used for marking the factored features in superclass. Also special syntax is used for method and parameter unification: one subclass method has the same name as the factored method *paint*, but the other has a different name *update*; parameter *Canvas canvas* is unified with *Canvas c*, which is not at the same position in class Ellipse.

Discussion About the Implementation of Signature Adaptation The set of transformations that deals with signature adaptation, corresponds to these atomic features: i) method/parameter renaming; ii) parameter addition/removal/reordering; iii) return type of method/parameter type changing.

To decide weather the name chosen for one method of the superclass may be propagated to the name of one subclass according to the example above is not straightforward. In particular, to rename the method in a subclass implies to parse all method calls in order to update it according to the new name. Moreover this may lead to name conflicts with other existing method names.

Furthermore about parameters we may consider renaming, addition and removal. To rename parameters in a method implies changing all the references to these parameters within the same method. Like for method names, name conflicts may arise if members and parameters have now the same name.

To add a new unused parameter will not yield any modification of the method code, but it may interfere with the lookup mechanism and the name of the new parameter could also introduce a conflict with members or other parameters.

To remove an unused parameter implies the identification of the code parts which depend on it, and to evaluate the side effects which are caused by the removal of that code. At first glance, it does not seem quite reasonable.

To reorder parameters within a method is orthogonal according to the code located in the body of that method. But it is obvious that heir or client classes which use the method, will be affected.

If a return type of a method or the type of a parameter is changed then it must conform to the original type, but it is not sufficient because the modification may interfere with the lookup and generate conflicts with existing methods[¶]. About type conformance, the rules proposed for handling primitive types should be the one found in the literature (e.g. in Java *double* can replace *float*). When the type deals with classes then a subtype conforms to the its supertype. But there is also another constraint: clients should use only the feature of the supertype.

Possible Implementation in Java According to the discussion made in previous paragraph, we propose a possible implementation solution which keeps the interface of the class intact and adds delegated methods with different names:

[¶]Moreover in Java the redefinition is non-variant and it is not possible to have two methods with the same name and parameters but with different return types.

```
class Ellipse {
    void update(double x, double y, Canvas canvas) {/* ellipse implementation */}
    void paint(Canvas c, int x, int y) { update(x,y,c); }}
```

In the implementation solution, class *Ellipse* keeps it's *update(...)* method intact and a new method *paint(...)* is added to perform factorization. The new added method *paint(...)* will be used as a delegated method which calls the *update(...)* method using the appropriate order of parameters. Of course it will be necessary to check that there is no name conflict.

Again, code transformation is used only for an implementation purpose. Those we are using, are a set of rules which translates a class hierarchy restructured using reverse inheritance class relationship, into a hierarchy having only normal, direct inheritance. The set of transformations deals with the functionalities discussed in previous paragraph.

5. CONCLUSIONS AND FUTURE WORK

In this paper we studied a possible semantics of the factorization mechanism in the framework of reverse inheritance relationship including signature adaptation and matching. We did not address the impact of reverse inheritance on some Java constructs. For example, we did not point out the semantics of the factorization according to the modifiers of methods or attributes.

Moreover we did not describe neither the semantics nor the implementation when reverse inheritance deals with interfaces or inner classes. Even if the analysis made in this paper is not complete, it suggests that reverse inheritance may be an interesting approach which may improve the reusability and the evolution capabilities of hierarchies of classes. In the near future we aim to finish the definition of the semantics and to validate it by an implementation as a plugin of Eclipse.

Even if it is far to be our first issue, extensions of Java or more generally of any object-oriented language with reverse inheritance may also be used in the context of the reorganization of hierarchy of classes as it has been suggested in 1. In this case it will only play the role of a specification language.

REFERENCES

1. Ciprian-Bogdan Chirila, Pierre Crescenzo, and Philippe Lahire. Towards reengineering: An approach based on reverse inheritance. Application to Java. Research report, Laboratoire Informatique, Signaux et Systemes de Sophia-Antipolis (UNSA / CNRS), France, July 2003.
2. Michel Dao, Marianne Huchard, Therese Libourel, and Cyril Roume. Evaluating and optimizing factorization in inheritance hierarchies. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
3. Peter H. Frohlich. Inheritance decomposed. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
4. Ted Lawson, Christine Hollinshead, and Munib Qutaishat. The potential for reverse type inheritance in Eiffel. In *Technology of Object-Oriented Languages and Systems (TOOLS'94)*, 1994.
5. William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring, 1993.
6. C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 407–417. ACM Press, 1989.
7. Markku Sakkinen. Exheritance - Class Generalization Revived. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
8. Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.